OCL-POLYHOK: PROGRAMANDO GPUS COM ELIXIR

HENRIQUE GABRIEL RODRIGUES[1]; ANDRÉ RAUBER DU BOIS[2]

[1]*Universidade Federal de Pelotas (UFPel) – henrique.grdr@inf.ufpel.edu.br*
[2]*Universidade Federal de Pelotas (UFPel) – dubois@inf.ufpel.edu.br*

1. INTRODUÇÃO

Este trabalho aborda a simplificação da programação de Unidades de Processamento Gráfico (GPUs) para computação de propósito geral (GPGPU), uma área que, apesar do alto poder de processamento paralelo das GPUs, é notoriamente complexa e de difícil adoção em larga escala. As dificuldades incluem a necessidade de conhecimentos em linguagens de baixo nível como C e C++, o gerenciamento de memória entre a CPU e a GPU, a sincronização entre os dispositivos e a familiaridade com a arquitetura única das GPUs – que está sempre em constante evolução (YANG; ZHANG; PAN, 2023).

A *Domain-Specific Language* (DSL) PolyHok (DU BOIS; CAVALHEIRO, 2025) foi criada para abstrair essa complexidade. permitindo que desenvolvedores foquem na lógica de execução paralela sem precisar lidar com tarefas de baixo nível. Desenvolvida para a linguagem funcional Elixir, a PolyHok explora os recursos de metaprogramação da linguagem para gerar, em tempo de execução, kernels CUDA semanticamente equivalentes ao código de alto nível escrito pelo programador. Com isso, a DSL abstrai tarefas de baixo nível como tipagem estática, alocação e desalocação de memória e transferências de dados entre host e device. A PolyHok também introduz um conceito inovador: kernels de alta ordem.

Kernels de alta ordem permitem que funções *device* sejam passadas como parâmetros para kernels, possibilitando a implementação elegante de padrões de paralelismo clássicos, como *map*, *reduce* e *scan*. Essa abordagem combina o desempenho de execução em GPU com a expressividade do paradigma funcional, oferecendo maior flexibilidade em comparação a frameworks tradicionais que impõem restrições rígidas no estilo de programação.

No entanto, a utilização de CUDA como *backend* para a PolyHok apresenta uma desvantagem significativa: portabilidade restrita. Por ser uma tecnologia proprietária da NVIDIA, a linguagem CUDA só pode ser utilizado em GPUs da própria empresa. Isso exclui severamente potenciais usuários da DSL com hardware de outros fabricantes, como Intel e AMD, além de diminuir o impacto da PolyHok na sua proposta de facilitar o uso de GPUs para programação de propósito geral.

Para superar essa limitação, o presente trabalho propõe a criação da **OCL-PolyHok**, uma implementação da PolyHok que utiliza OpenCL (*Open Computing Language*) no *backend* em vez de CUDA. Diferentemente do CUDA, o OpenCL é um padrão aberto e multiplataforma para programação heterogênea, mantido pelo consórcio Khronos Group, que reúne empresas como Intel, AMD, Apple e a própria NVIDIA (MUNSHI et al., 2011). Com o OpenCL, a DSL PolyHok poderá ser executada em praticamente qualquer dispositivo heterogêneo, composto por CPUs, GPUs e FPGAs de qualquer fabricante.

2. METODOLOGIA

Inicialmente, uma análise detalhada da implementação atual da PolyHok foi realizada, buscando entender como era realizada a tradução do código da DSL para CUDA, e como esse código era compilado e executado na GPU. Esse estudo forneceu o conhecimento necessário para compreender a arquitetura da DSL. Com isso, pôde-se determinar quais pontos deveriam ser modificados e reestruturados para a PolyHok utilizar OpenCL.

Em seguida, foi realizada uma revisão bibliográfica sobre CUDA e OpenCL, com ênfase nas diferenças e semelhanças entre os dois modelos de programação heterogênea. Obras clássicas como *CUDA by Example* (SANDERS; KANDROT, 2010) e *OpenCL Programming Guide* (MUNSHI et al., 2011) forneceram subsídios teóricos para o entendimento das particularidades de cada abordagem. Essa etapa foi fundamental para a correta modificação do código da PolyHok, garantindo que a implementação baseada em CUDA fosse corretamente reescrita em OpenCL, mantendo não só a funcionalidade, como também a eficiência.

A fase seguinte consistiu na implementação do novo *backend*. Este foi implementado utilizando a linguagem C++ com o auxílio da biblioteca NIFs (*Native Implemented Functions*), responsável por fazer a interface entre o código C++ (nativo) e o código Elixir. A linguagem C++ foi escolhida por dispor de recursos de programação orientada a objetos que facilitaram significativamente a construção do *backend*, como classes auxiliares e o *wrapper* de C++ do OpenCL (*opencl.hpp*) que possui classes representando objetos OpenCL extremamente simples de manipular e gerenciar.

Após a implementação, planeja-se realizar uma série de testes e benchmarks para avaliar a corretude e o desempenho da OCL-PolyHok em comparação com a versão original baseada em CUDA e uma versão puramente em Elixir dos testes. Até o momento, a implementação ainda não foi submetida a estes benchmarks, apenas testes preliminares foram realizados.

O projeto está disponível em um repositório público no GitHub[1], facilitando o acesso da comunidade acadêmica e de desenvolvedores para que possam utilizá-la e contribuir para seu aprimoramento.

3. RESULTADOS E DISCUSSÃO

Até o momento, foi concluído o estudo da arquitetura da PolyHok e a primeira versão do *backend* em OpenCL foi implementada. Essa versão já permite a execução de kernels simples em dispositivos não-NVIDIA escritos em OCL-PolyHok.

O Código 1 apresenta o módulo *ArraySum* onde definiu-se um kernel em OCL-PolyHok para calcular a soma de dois *arrays* na GPU. No módulo também há a função *sum/2*, responsável por executar o kernel e armazenar o resultado em um novo vetor *gnx* na GPU. Este kernel, junto com outros códigos experimentais, foram executados em uma máquina com uma CPU AMD Ryzen 5 5500U, equipada com a GPU integrada AMD *gfx90c*, no sistema operacional Linux Mint 22.1 Xia.

[1] Disponível em: < https://github.com/Equiel-1703/ocl-polyhok>.

```
require OCLPolyHok

OCLPolyHok.defmodule ArraySum do
  # Kernel que calcula a soma
  defk sum_ker(a1, a2, result_array, size) do
    index = blockIdx.x * blockDim.x + threadIdx.x
    stride = blockDim.x * gridDim.x

    for i in range(index, size, stride) do
      result_array[i] = a1[i] + a2[i]
    end
  end

  # Função Elixir que invoca o kernel na GPU
  def sum(input_1, input_2) do
    shape = OCLPolyHok.get_shape(input_1)
    type = OCLPolyHok.get_type(input_1)

    # Criando um array na GPU para armazenar o resultado
    result_gpu = OCLPolyHok.new_gnx(shape, type)


    size = Tuple.product(shape)
    threadsPerBlock = 128
    numberOfBlocks = div(size + threadsPerBlock - 1, threadsPerBlock)


    # Executando o kernel
    OCLPolyHok.spawn(&ArraySum.sum_ker/4,
              {numberOfBlocks, 1, 1},
              {threadsPerBlock, 1, 1},
              [input_1, input_2, result_gpu, size]) # Argumentos do kernel

    # Retorna o array na GPU contendo o resultado
    result_gpu
  end

end

a = Nx.tensor(Enum.to_list(1..100), type: {:f, 32}) |> OCLPolyHok.new_gnx
b = Nx.tensor(Enum.to_list(1..100), type: {:f, 32}) |> OCLPolyHok.new_gnx

result = ArraySum.sum(a, b) |> OCLPolyHok.get_gnx

IO.inspect(result, label: "Resultado da soma: ")
```

Código 1 – Código em Elixir usando a DSL OCL-PolyHok para somar dois *arrays*.

Os testes preliminares indicam que a tradução das instruções de alto nível para OpenCL está sendo feito de maneira correta, preservando a semântica. Os

testes também mostram que a memória na GPU está sendo alocada e desalocada corretamente através do coletor de lixo da máquina virtual do Erlang, o acesso a índices de memória está sendo traduzido corretamente usando a sintaxe OpenCL e os kernels estão sendo compilados e executados sem nenhum erro.

Ainda é necessário realizar testes para verificar a funcionalidade dos kernels de alta ordem, recebendo funções anônimas e funções *device* como argumento, e também verificar o desempenho do *backend* OpenCL em comparação com o *backend* em CUDA, através de *benchmarks*.

4. CONCLUSÕES

O desenvolvimento da OCL-PolyHok representa uma contribuição relevante para a área de GPGPU ao eliminar a dependência exclusiva de GPUs NVIDIA da PolyHok, ampliando sua compatibilidade para diferentes arquiteturas heterogêneas.

Embora os testes com kernels de alta ordem e de desempenho ainda estejam em andamento, os resultados preliminares mostram que a OCL-PolyHok já atingiu um de seus principais objetivos: portabilidade. Assim, este projeto contribui para tornar a programação em GPUs mais acessível e prática, abrindo caminho para estudos futuros de otimização e avaliação de desempenho em cenários mais complexos.

5. REFERÊNCIAS BIBLIOGRÁFICAS

DU BOIS, A. R.; CAVALHEIRO, G. **Polymorphic Higher-Order GPU Kernels**. Proceedings of the 31st International European Conference on Parallel and Distributed Computing. **Proceedings**...Springer Nature, 2025.

MUNSHI, A. et al. **OpenCL Programming Guide**. 1. ed. Boston, MA: Addison-Wesley Professional, 2011.

SANDERS, J.; KANDROT, E. **CUDA by Example**. Boston, MA: Addison-Wesley Professional, 2010.

YANG, W.; ZHANG, C.; PAN, M. Understanding the Topics and Challenges of GPU Programming by Classifying and Analyzing Stack Overflow Posts. **Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering**, p. 1444–1456, 30 nov. 2023.